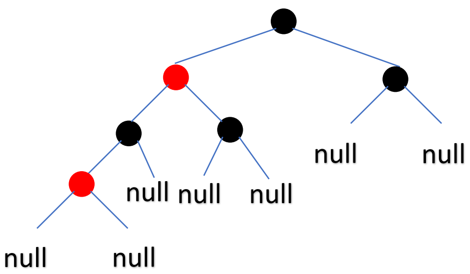
Algorithm: Lab8 (By Sujiv Shrestha ID:610145)
Problem 1.

1. An *AVL Tree* is a BST that satisfies a different balance condition, namely: The AVL Balance Condition For each internal node x, the height of the left child of x differs from the height of the right child of x by at most 1. (Equivalently, the heights of the left and right subtrees of x differ by at most 1.)

Create a red-black tree that does *not* satisfy the AVL Balance Condition.

Solution:

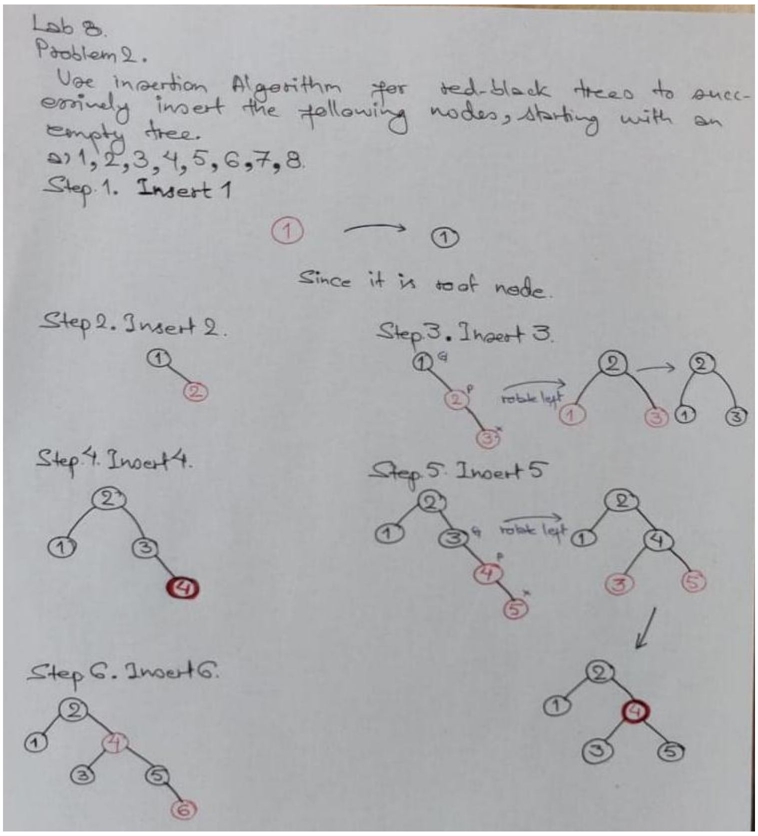


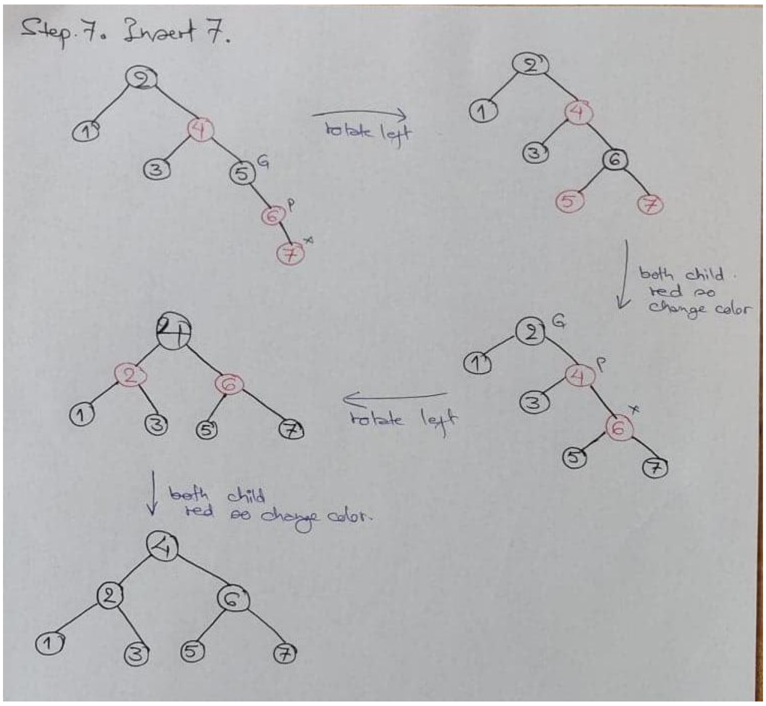
For the root node of this Red-Black tree the height of the left child is 2 while the height of the right child is 0. So, this Red-Black tree does not satisfy the AVL Balance Condition.

Problem2

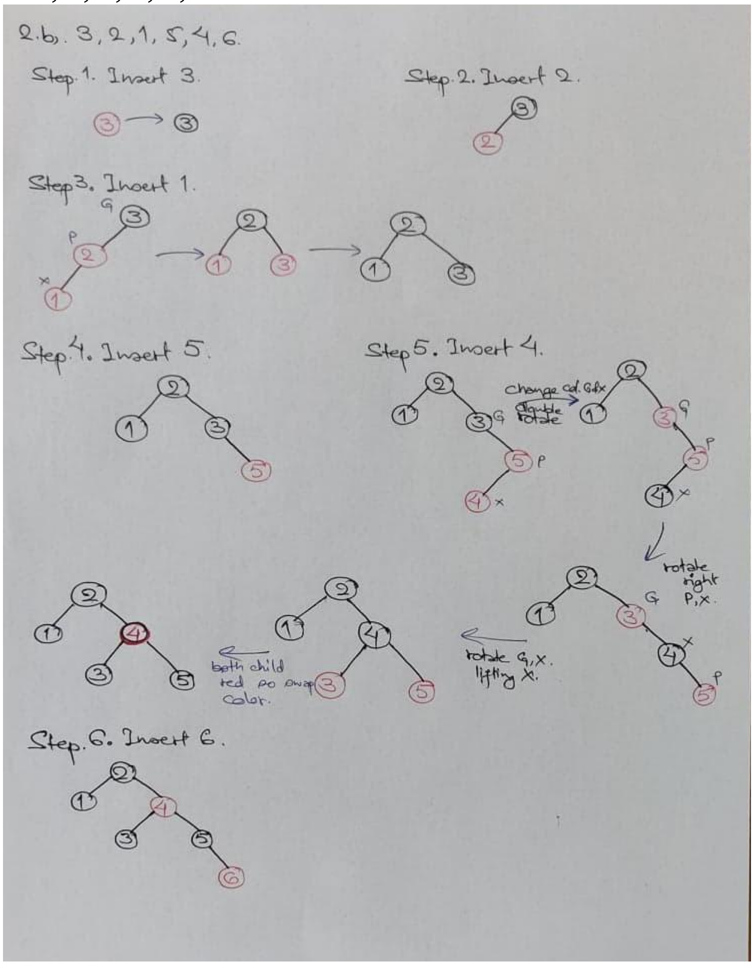
2. Use the insertion algorithm for red-black trees to successively insert the following nodes, starting with an empty tree.

a. 1, 2, 3, 4, 5, 6, 7, 8




Step.7. Insert 7.
rotate left
both child red so change color
rotate left
both child red so change color.

b. 3, 2, 1, 5, 4, 6


2.b). 3, 2, 1, 5, 4, 6.
Step.1. Insert 3.
Step.2. Insert 2.
Step3. Insert 1.
Step.4. Insert 5.
Step5. Insert 4.
rotate right P,X.
rotate G,X. lifting X.
both child red so swap color.
Step.6. Insert 6.

Note on Part (a): Recall that an already sorted insertion sequence is a worst case for an ordinary BST. Notice how the red-black balancing operations handle this to remain balanced.

Yes, it somehow tries to restructure the unbalanced BST to balanced even in the worst case.

Problem3

3. Devise an algorithm IsPrime(n) which outputs TRUE if n is prime, FALSE otherwise. Then implement as a Java method. What is the asymptotic running time of IsPrime? Explain.

Algorithm isPrime(n)
    Input integer n to check if it is prime
    Output Boolean: true if it is prime number and false if it is not
    for i←2 to √n **do**
        if(n%i=0) **then**
            return false
    return true

Java Implementation:

```
public static boolean isPrime(int n) {
        for(int i=2;i<=Math.sqrt(n);i++) {
                if(n%i==0) {
                        //Divisible by i
                        return false;
                }
        }
        return true;
}
```

Running Time, $T(n) = O(\sqrt{n})$

Problem4

4. In the course, we have determined asymptotic running times of sorting algorithms as a function of input size *n*. However, in number-theoretic algorithms, such as GCD, the running time has been bounded by functions of *n* where, in this case, *n* is an input *value*, but does not represent the input *size*. The reason is that the *size* of a natural number *n*, from the point of view of any reasonable computational model, is *its length as a bit string* and not simply the value *n* itself.
Examples:
If n = 7, its size as the bit string 111 is 3.
If n = 67, its size as the bit string 1000011 is 7.
In general, $length(n) = \lfloor \log n \rfloor + 1$.
When running times of number-theoretic algorithms are expressed in terms of input *size* rather than input *value* (as we have done so far), results can appear unfamiliar.
For example, the asymptotic running time of GCD(*m*,*n*) in terms of input values, as we have seen, is O(log *n*). However, since *n* is O(2*length*(*n*)), in terms of input *size*, GCD(*m*,*n*) runs in O(*length*(*n*)). That is, GCD(*m*,*n*) is *linear in the size of n*. Here is a more careful analysis:
GCD Algorithm
Algorithm GCD(m,n)
    Input nonnegative integers m, n, not both 0
    Output gcd(m,n)
    if n=0 **then**
        return m

else

return GCD(n, m % n)

(**) In general, if T(n) is O(f(n)), in terms of the *value n,* then, in terms of the size b = b(n) of input n, T(b) is O(f(2^b)). In the case of GCD, since gcd(m,n) runs in O(log n), in terms of the value n, gcd runs in O(log 2^b) = O(b) in terms of input size.
In light of the above discussion, answer the following:

A. Express the asymptotic running time of your algorithm IsPrime(n) in terms of the input *size* rather than input value. It may be helpful to use two arguments, n, $b(n)$, to help focus on the number of bits of n when computing running time; then you can compute running time $T(b)$ in terms of the input size. Or you can simply compute the running time in terms of n, then convert to running time in terms of b using the formula given in (**) above.
Solution:

Running Time of my isPrime algorithm in terms of value of input, T(n) = O(√n)
Or, T(n) is O(n)
The size of the bit stream representing number n is, b = length(n) = ⌊logn⌋+1

Since, b = length(n) = ⌊logn⌋+1
or, n ≥ 2^{b-1}
so, we can also say that T(b) is O(2^{b-1})

B. Suppose $T(b)$ is the running time of your algorithm in terms of input size. Show that b^2 is o($T(b)$).
Solution:
Here,

We have T(b) is the running time of my algorithm in terms of input size,
where b = length(n)
b = ⌊logn⌋+1
b-1 ≤ logn
taking anti-log (base-2) on both sides, $2^{b-1} \le n$

we have T(b) is O(2^{b-1}), so if b^2 is o(2^{b-1}) then b^2 is o(T(b))
Hence, lets prove b^2 is o(2^{b-1})

Calculate: $\lim_{b\to\infty} \frac{b^2}{2^{b-1}}$

$= \lim_{b\to\infty} \frac{2b}{2^{b-1}log2}$ [∵ Applying L. Hopital Rule]

$= \lim_{b\to\infty} \frac{2}{2^{b-1}*2^{b-1}*log2*log2}$ [∵ Applying L. Hopital Rule for second time]

= 0

So, we get b^2 is o(2^{b-1})
or, b^2 is o(T(b))